Corrigé TP12

0. Le nombre minimal de boîtes est 3 et ces dernières sont toutes complètement remplies (l'une avec les volumes 2 et 8, la deuxième avec les volumes $1, 4, 5$ et la la dernière avec les volumes 7 et 3).

1. On peut par exemple proposer le type suivant :

```
type box = {remaining_volume : int;
            elements : int list}
```

   Le premier champ stocke le volume laissé libre dans la boîte. Le second champ stocke le numéro des objets dans la boîte, pour en ajouter un il suffira de faire une adjonction en tête de liste. Les deux opérations décrites par l'énoncé peuvent ainsi se faire en temps constant.

   On peut rendre ces champs mutables pour modifier les boîtes plutôt que de les écraser par une nouvelle boîte dans les algorithmes ci-dessous. Faites le en exercice et demandez vous si ça change quelque chose en termes de complexité.

2. La boîte courante sera évidemment celle en tête de liste. Si cette dernière peut accueillir l'objet, on l'y place en modifiant le volume restant non occupé dans ladite boîte. Sinon, on créé une nouvelle boîte en tête de liste dans laquelle placer l'objet (et il rentrera forcément dans une boîte vide puisqu'on a supposé que les volumes des objets étaient tous inférieurs au volume d'une boîte).

```
let add_next (i:int) (v:int) (l:box list) (v_max:int) :box list =
  match l with
    |[] -> [{remaining_volume = v_max - v; elements = [i]}]
    |t::q -> let rv = t.remaining_volume and elems = t.elements in
             if rv >= v then {remaining_volume = rv - v ; elements = i::elems}::q
             else {remaining_volume = v_max - v; elements = [i]}::t::q
```

   Remarque 1 : Attention à suivre scrupuleusement l'heuristique décrite : dès qu'on rencontre un objet qui ne rentre pas dans la boîte courante, on n'ajoutera plus jamais d'objet dans celle-ci, même si certains pourraient y rentrer.

   Remarque 2 : On pourrait en fait se passer du cas où la liste est vide en fusionnant ce cas avec celui où l'objet ne rentre pas dans la première boîte : les deux se traitent de la même façon.

3. Il suffit d'ajouter chaque objet dans la liste de boîtes à l'aide de `add_next` et de déterminer après ces ajouts la longueur de la liste de boîtes.

```
let next_fit (v_max:int) (volumes:int array) :int =
  let boxes = ref [] in
  for i = 0 to (Array.length volumes) -1 do
    boxes := add_next i volumes.(i) !boxes v_max
  done;
  List.length !boxes
```

4. On parcourt la liste de boîtes jusqu'à en trouver une qui peut accueillir l'objet considéré. Si aucune ne convient, on créé une nouvelle boîte où placer l'objet via le cas de base correspondant à un ajout dans une liste vide de boîtes.

```
let rec add_object (i:int) (v:int) (l:box list) (v_max:int) :box list =
  match l with
  |[] -> [{remaining_volume = v_max - v; elements = [i]}]
  |t::q -> let rv = t.remaining_volume and elems = t.elements in
           if rv >= v then
             {remaining_volume = rv - v; elements = i::elems}::q
           else
             t::(add_object i v q v_max)
```

5. On peut reprendre exactement la fonction de la question 3 en modifiant le mode d'ajout. Pour changer, on propose une version plus fonctionnelle du remplissage des boîtes.

```
let first_fit (v_max:int) (volumes:int array) :int =
  let rec add_multiple_objects (i:int) (current_packs:box list) :box list =
    if i >= Array.length volumes then current_packs
    else
      add_multiple_objects (i+1) (add_object i volumes.(i) current_packs v_max)
  in List.length (add_multiple_objects 0 [])
```

6. Comme l'utilisation de Array.sort est permise, cette fonction est immédiate.

```
let ffd (v_max:int) (volumes:int array) :int =
  Array.sort (fun x y -> y-x) volumes;
  first_fit v_max volumes
```

   On rappelle que Array.sort prend en entrée une fonction de comparaison et un tableau et modifie ce tableau en le triant selon l'ordre décrit par la fonction de comparaison. La fonction de comparaison doit renvoyer 0 si ses deux arguments sont égaux, une valeur strictement positive si le premier est strictement plus grand que le second selon l'ordre considéré et strictement négative sinon. Attention au fait qu'on souhaite trier les volumes dans l'ordre décroissant.

7. On obtient les répartitions suivantes sur l'exemple de la question 0 :

	Numéro des objets dans chaque boîte	Nombre de boîtes
next-fit	$\{0,1\},\{2\},\{3,4\},\{5\},\{6\}$	5
first-fit	$\{0,1,4\},\{2,5\},\{3\},\{6\}$	4
FFD	$\{0,5\},\{1,4\},\{2,3,6\}$	3

   Dans la dernière ligne, on a renuméroté les objets de sorte à ce que le premier ait le plus gros volume, mais on pourrait facilement retrouver les numéros d'origine. En modifiant nos trois approximations pour observer la liste de boîtes calculées, on constate qu'on obtient les mêmes répartitions.

   Si on note $\alpha_{nf}, \alpha_{ff}, \alpha_{ffd}$ les facteurs d'approximation de next-fit, first-fit et FFD (en admettant que ce sont bien des algorithmes d'approximation à facteur constant pour BIN PACKING, ce qu'on montrera en constatant que next-fit en est un dans la deuxième partie), on peut avoir l'intuition que $\alpha_{ffd} \leq \alpha_{ff} \leq \alpha_{nf}$. On peut confirmer cette intuition en appliquant ces fonction à de nouvelles entrées générées via random_volumes qui renvoie un tableau de n volumes choisis aléatoirement entre 1 et un volume maximal v_max.

```
Random.self_init();;

let random_volumes (n:int) (v_max:int) :int array =
  let volumes = Array.make n 0 in
  for i = 0 to n-1 do
    volumes.(i) <- (Random.int v_max) +1
  done;
  volumes
```

8. Le problème BPD consiste à se demander s'il est possible de ranger les n objets dans un nombre de boîtes fixé sans en dépasser la capacité :

$$\begin{cases} \textbf{Entrée} : v_1, ..., v_n \in \mathbb{N}^\star, V \in \mathbb{N}^\star \text{ et } m \in \mathbb{N}. \\ \textbf{Question} : \text{Y a-t-il une fonction } f : [\![1, n]\!] \to [\![1, m]\!] \text{ telle que pour tout } i \in [\![1, m]\!], \sum_{f(v_j)=i} v_j \leq V \ ? \end{cases}$$

9. Si $v_1, ..., v_n, V, m$ est une instance de BPD, sa taille est de l'ordre de $n \times \max(\log v_i) + \log m + \log V$ si on ne néglige pas la taille des entiers impliqués.

   Etant donnés une instance de BPD et un tableau de taille n contenant en case i un entier inférieur à n (ce qui est loisible car au pire on place chacun des n objets dans une boîte ; la taille de ce tableau est ainsi en $O(n \log n)$ donc polynomiale en la taille de l'instance) correspondant au numéro de la boîte dans laquelle ranger i, on peut vérifier que ce tableau encode un rangement correct en temps polynomial en la taille de l'instance comme suit :

   - On calcule le poids de chacune des au plus n boîtes en parcourant le tableau, ce qui peut se faire en $O(n \max(\log v_i))$ puisqu'on va faire au plus n sommes d'entiers de taille inférieure à $\max(\log v_i)$ et on stocke ces poids.
   - Pour chacun de ces au plus n poids, on vérifie qu'ils sont inférieurs à V.
   - On vérifie qu'il y a moins de m boîtes impliquées ce qui implique une comparaison entre un entier de taille au plus $\log n$ et un entier de taille $\log m$. Là encore, cette opération se fait en temps polynomial en la taille de l'instance.

   Remarque : Je doute fortement qu'on vous demande d'être aussi pointilleux dans une preuve qu'un problème est dans NP. Signaler que vérifier si un tableau décrivant un rangement est correct peut se faire en temps polynomial en la taille de l'instance et que la taille d'un tel tableau est également polynomiale suffirait.

10. La transformation proposée peut clairement être effectuée en temps polynomial en la taille de l'instance de PARTITION puisqu'elle exige simplement de faire n sommes et n multiplications par une constante impliquant les c_i.

    Si $c_1, ..., c_n$ est une instance positive de PARTITION, il existe $I \subset [\![1, n]\!]$ tel que $\sum_{i \in I} c_i = \sum_{i \notin I} c_i$. Ainsi :

$$2V = \sum_{i=1}^{n} 2c_i = \sum_{i \in I} 2c_i + \sum_{i \notin I} 2c_i = 2\left(\sum_{i \in I} 2c_i\right) \text{ donc } \sum_{i \in I} 2c_i = \sum_{i \notin I} 2c_i = V$$

    avec V le volume des boîtes dans l'instance de BPD construite à partir de celle de PARTITION. Il est donc possible de placer les objets de poids $2c_1, ..., 2c_n$ dans exactement deux boîtes (complètement remplies) de volume V et en prime, le sous-ensemble I nous indique comment les répartir.

Réciproquement, supposons que $2c_1, ...2c_n, V = \sum_{i=1}^{n} c_i, 2$ est une instance positive de BPD. Alors il existe $I \subset [\![1, n]\!]$ tel que les objets dans la première boîte sont ceux indicés par I et ceux dans la deuxième ceux indicés par les éléments n'appartenant pas à I. Comme le volume total de tous les objets est égal à $2V$ et qu'il rentrent dans 2 boîtes de volume V alors le volume occupé par les objets de chacune des deux boîtes est exactement égal à V. En particulier ces deux volumes sont égaux donc

$$\sum_{i \in I} 2c_i = \sum_{i \notin I} 2c_i$$

ce qui montre que $c_1, ..., c_n$ est une instance positive de PARTITION après simplification par deux.

Comme PARTITION est NP-difficile (puisque NP-complet) et que PARTITION se réduit polynomialement en BPD d'après ce qui précède, on en déduit que BPD est NP-difficile. Comme il est aussi NP d'après la question 9, il est NP-complet.

11. Si on avait deux boîtes consécutives B_1 et B_2 (B_1 ayant été créée après B_2 après avoir rencontré un objet qui ne pouvait pas rentrer dans B_2) dont les volumes occupés v_1 et v_2 vérifiaient $v_1 + v_2 \leq V$ alors tous les objets de ces deux boîtes auraient pu rentrer dans B_2 et donc B_1 n'aurait pas été créée d'après le fonctionnement de l'heuristique next-fit. C'est une contradiction.

12. Le volume total V_T occupé par les objets dans les $m^\star$ boîtes est plus grand que le volume de l'ensemble des objets ; autrement dit, $V_T \geq \sum_{i=1}^{n} v_i$. Mais par ailleurs $m^\star V \geq V_T$ car au mieux, chacune des $m^\star$ boîtes est complètement remplie. On en déduit que $m^\star V \geq \sum_{i=1}^{n} v_i$.

13. Si on note $B_1, ..., B_m$ les m boîtes déterminées par next-fit, en sommant sur les paires de boîtes consécutives et en utilisant la question 11 on a

$$\begin{aligned}
(m-1)V &< \sum_{i=1}^{m-1} (\text{volume occupé dans } B_i + \text{ volume occupé dans } B_{i+1}) \\
&\leq 2 \times \sum_{i=1}^{m} (\text{volume occupé dans } B_i) \\
&= 2 \times \sum_{i=1}^{n} v_i \quad \text{car le volume total occupé est égal au volume total des objets} \\
&\leq 2m^\star V \quad \text{d'après la question 12}
\end{aligned}$$

On en déduit que $m < 2m^\star + 1$ et toutes ces quantités étant entières, $m \leq 2m^\star$ ce qui conclut.

14. L'ajout d'un nouvel objet dans la liste de boîtes en construction selon la stratégie next-fit peut se faire en temps constant. On traite en tout n objets d'où une complexité en $O(n)$. On vient donc de troquer un algorithme a priori exponentiel permettant de résoudre BIN PACKING de manière exacte pour un algorithme polynomial et même linéaire qui calcule une solution, certes approchée, mais avec la garantie que cette dernière est au pire 2 fois plus grande que l'optimale.

15. Non. Pour tout $n \in \mathbb{N}$, on considère l'instance I_n de BIN PACKING suivante : le volume V vaut n et on souhaite ranger $2n$ objets dont les volumes sont donnés par la suite $n, 1, n, 1, ..., n, 1$. Dans ces conditions, next-fit utilise $C_n = 2n$ boîtes en plaçant un objet dans chacune alors que la solution optimale consiste à remplir n boîtes avec les objets de volume n et une boîte avec les n objets de volume 1 pour un total de $C_n^\star = n + 1$ boîtes.

Comme $C_n/C_n^\star \to 2$ lorsque $n \to +\infty$, cela garantit qu'on ne peut pas trouver de meilleur facteur d'approximation que 2 pour next-fit.

16. On note $\mathcal{P}$ la $(3/2-\varepsilon)$-approximation dont l'existence est supposée par l'énoncé.

    Soit $c_1, ..., c_n$ une instance de PARTITION. On considère l'instance I de BIN PACKING $2c_1, ...2c_n, V = \sum_{i=1}^{n}, 2$. Si $c_1, ..., c_n$ est une instance positive de PARTITION alors le nombre optimal de boîtes pour I est 2 et le nombre de boîtes m renvoyé par $\mathcal{P}$ sur l'instance I vérifiera donc $m \leq 2 \times (3/2-\varepsilon) = 3-2\varepsilon$. Puisque m est un entier, on aura ainsi $m = 2$. Si $c_1, ..., c_n$ n'est pas une instance positive de PARTITION, alors l'algorithme $\mathcal{P}$ appliqué à I renverra au moins 3.

    Ainsi, pour résoudre PARTITION polynomialement, il suffit de construire l'instance de BIN PACKING correspondante, ce qui se fait polynomialement comme vu en question 10 puis de lui appliquer $\mathcal{P}$, qui est lui aussi polynomial : si le résultat de cet algorithme est 2 alors l'instance considérée de PARTITION est positive, sinon, elle est négative.